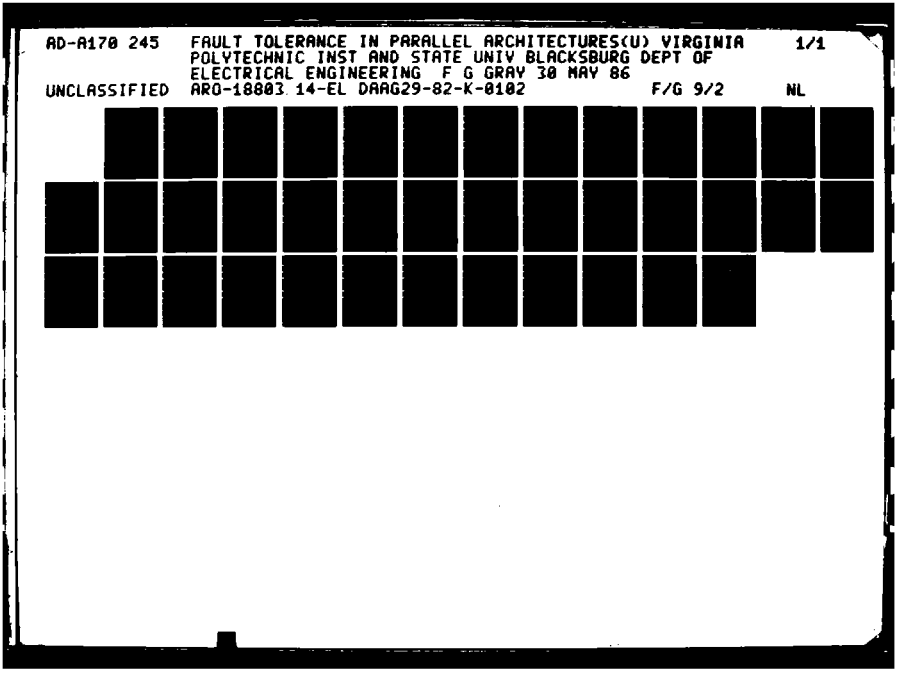
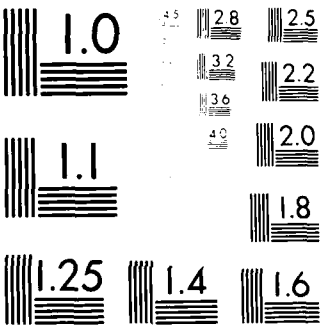

AD-A170 245 FAULT TOLERANCE IN PARALLEL ARCHITECTURES(U) VIRGINIA 1/1
POLYTECHNIC INST AND STATE UNIV BLACKSBURG DEPT OF
ELECTRICAL ENGINEERING F G GRAY 30 MAY 86
UNCLASSIFIED ARO-18803.14-EL DAAG29-82-K-0102 F/G 9/2 NL



FAULT TOLERANCE IN PARALLEL ARCHITECTURES FINAL REPORT

May 30, 1986

Dr. F. Gail Gray

Prepared for U.S. Army Research Office
Grant DAAG29-82-K-0102
Prepared by
Department of Electrical Engineering
Virginia Tech
Blacksburg, Virginia 24061

Abstract

This paper describes a proposed automatically reconfigurable cellular architecture. The unique feature of this architecture is that the reconfiguration control is distributed within the system. There is no need for global broadcasting of switch settings. This reduces the interconnection complexity and the length of data paths. The system can reconfigure at the request of the applications software or in response to detected faults. This architecture supports fault tolerant applications since the reconfiguration can be self- triggered from within. The complete reconfiguration process can proceed without external interference.

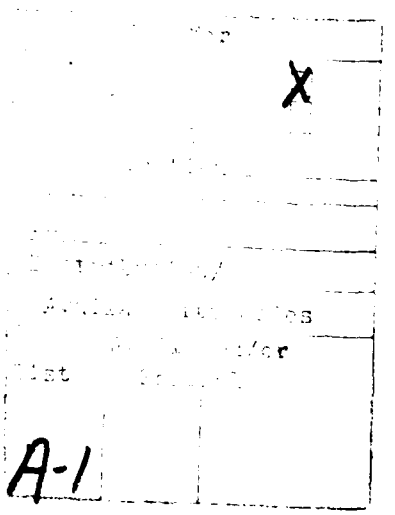


Table of Contents

1.0 Statement of the Problem **1**
1.1 Proposed Structure 1
1.2 Theory of Cellular Reconfiguration 2
1.3 References 6

2.0 Summary of Results **7**
2.1 Computation Hyperplane 7
2.2 Control Hyperplane 9
2.2.1 Patterns and Growth 10
2.2.1.1 Proof of Concept Example 10
2.2.1.2 Alignment of Master Pattern 15
2.2.1.3 Clearing the Array 15
2.2.2 Fault Tolerance 20
2.2.2.1 Determining the Size of Fault Free Space 20
2.2.2.2 Two Dimensional Reconfiguration Algorithm 21
2.2.2.3 One Dimensional Reconfiguration Algorithm 25
2.2.3 Input/Output Path Construction 25
2.3 System Concepts 27
2.3.1 Synchronized Clocks 27
2.3.2 Simulation 29
2.3.3 Periodic Self Restoration 29
2.3.4 Cell Testing Techniques 30
2.4 References 30

3.0 Publications and Technical Reports **31**

4.0 Participating Scientific Personnel **33**

1.0 Statement of the Problem

Recent advances in VLSI technology has made it possible to interconnect many small computers together to achieve high parallelism. There are many topologies for interconnecting processors. The optimum choice for interconnection strategy is frequently application dependent; therefore, most of these architectures can be used only for a small number of applications.

Kung [1] has proposed such a class of architectures known as systolic arrays that can be used to perform a variety of highly parallel computations such as matrix multiplication, fast Fourier transformation, etc. Each processor in these arrays performs a simple and short computation and regularly pumps data in and out. But only a limited number of functions can be performed with each type of interconnection network. What is needed is a general purpose reconfigurable architecture that allows many of the special purpose architectures to be embedded in a single structure.

Considerable attention is being given to such a general purpose reconfigurable architecture in recent literature. Snyder [2] demonstrated the feasibility of such an architecture with the CHIP computer (Configurable, HIghly Parallel computer), which provides a programmable interconnection structure integrated with processing elements. It is designed to provide the flexibility needed to compose general solutions while retaining the benefits of uniformity and locality that the algorithmically specialized processors exploit. It consists of a switch lattice, in which the switches are set to create the best interconnection network for the function to be executed. An external controller broadcasts a command to all the switches to invoke the appropriate architecture.

This approach has two major disadvantages. First, the setting of the switches is controlled by an external processor. This necessitates some type of global connection to all switches. Secondly, the master control circuitry becomes a single point failure site, since it would be necessary for the master control to work correctly in order for the appropriate switch setting to be invoked. Thus the existence of the master control will significantly degrade the system reliability. This is highly undesirable for failure critical applications such as automatic landing of commercial aircraft, control systems for nuclear reactors, life support systems for medical applications, etc. Even in less critical applications, system downtime is often very expensive.

In this research project, we investigated a reconfigurable cellular architecture in which the reconfiguration mechanism is distributed throughout the array instead of being a single-point failure problem. In addition, reconfiguration does not require an external processor to compute the new interconnection pattern.

1.1 Proposed Structure

Our desire is to be able to implement a set of specific architectures designed to solve a specified set of parallel computations in a single reconfigurable system. Cellular arrays are a viable computational architecture for such an implementation. A cellular (iterative) array is a collection of identical cells that are interconnected in a uniform, or regular, fashion. A cellular array is proposed for the following reasons. First, they are of highly parallel nature. Secondly, since all the cells in the array are identical the architecture becomes easily expandable without changing the current hardware in any significant way. Lastly, each processor is connected to other processors according to a regular interconnection pattern. By using regular local interconnection patterns we can avoid the use of global connections, so that the interconnection complexity will not increase with the size of the system. By making the control distributed throughout the array, the "hard core" component

will be minimal. Being a planar, regular structure, such a parallel processor is well suited for VLSI implementation.

The proposed cellular structure is composed of two cellular arrays that are interconnected as shown in Figure 1 on page 3. The cellular structure consists of a "control hyperplane" and a "computation hyperplane", where for each cell in the computation hyperplane there is an associated cell in the control hyperplane. Each cell in the computation hyperplane can be either a switch or a processing element, as shown in Figure 2 on page 4. If it is a processing element it must be complex enough to realize the functions required by each of the algorithms, i.e., each processing element is some type of universal logic module, or microprocessor, that can perform a list of functions.

Each cell in the computation plane is controlled by the state of the corresponding cell in the control plane. If the cell in the computation plane is a processing element, then the control cell specifies a particular algorithm from a set of possible algorithms that the processing element can implement. If the cell in the computation plane is a switching element, then the control cell will specify a particular local interconnection of the switching element to its neighbors. The overall function to be performed by the cellular structure is defined by the global pattern of control states.

To create the desired configuration for a particular computation, one cell in the array is initialized to a "seed" state which defines the global computational task to be performed by the array. The cellular array will then utilize that information to "grow" the required pattern of states in the control hyperplane. The pattern of states in the control hyperplane invokes the desired interconnection structure in the computation plane.

1.2 Theory of Cellular Reconfiguration

The control hyperplane is responsible for assigning proper functions to the cells in the computational hyperplane. Any arbitrary cell in the control hyperplane will eventually receive information about the global function to be implemented from the "seed" state initially planted at an arbitrary location. The way in which the given information is distributed throughout the control plane to produce a desired final pattern is explained in this section. This process will be referred to as 'growth'.

The control hyperplane is an array of identical cells interconnected in a uniform fashion, where each such cell can receive state information only from its neighbors. The cell which initially receives information about the global function will communicate with its neighbors and gradually spread the information through the array to create the final desired pattern. This section explains the "growth" process, the way in which the given information is transmitted throughout the control plane produce a desired final pattern. Since the control hyperplane is an array of identical cells interconnected in a uniform fashion, each cell can receive state information only from its neighbors. This process is illustrated in the following example.

Cell		-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7		time
	...	0	0	0	0	0	0	0	S	0	0	0	0	0	0	0	...	0
	...	0	0	0	0	0	0	A	A	A	0	0	0	0	0	0	...	1
	...	0	0	0	0	0	B	B	C	B	B	0	0	0	0	0	...	2
	...	0	0	0	0	E	E	I	I	E	E	E	0	0	0	0	...	3
	...	0	0	0	C	O	M	P	U	T	E	R	S	0	0	0	...	4

Each row describes the state of the one-dimensional array at a particular time. At t = 0, the "seed" state "S" is planted at an appropriate place in the array with all other cells in the "quiscent" state "0". At each time step, each cell observes the state of each of its immediate neighbors and its own state, then decides what its next state will be.

For example, at time t = 0, cell 0 observes that the cell to its immediate left is in the "0" state, cell 0 is in the "S" state and the cell to its immediate right is in the "0" state. The *local pattern* for cell 0 at time 0 is said to be 0S0. When any cell in the array is in a local pattern of 0S0 at time t, it will change its own state to state A at time t + 1. Similarly, if a cell is in a local pattern of 00S

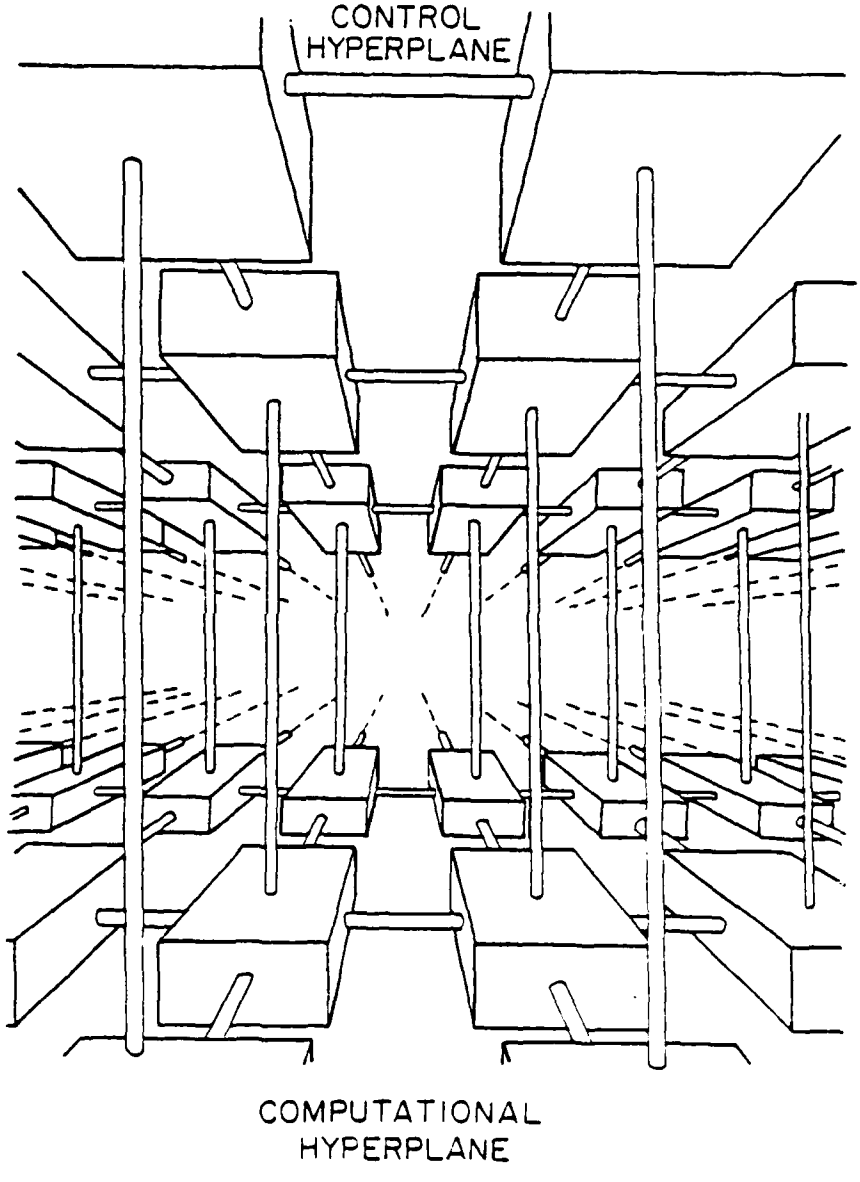


Figure 1. Cellular Architecture

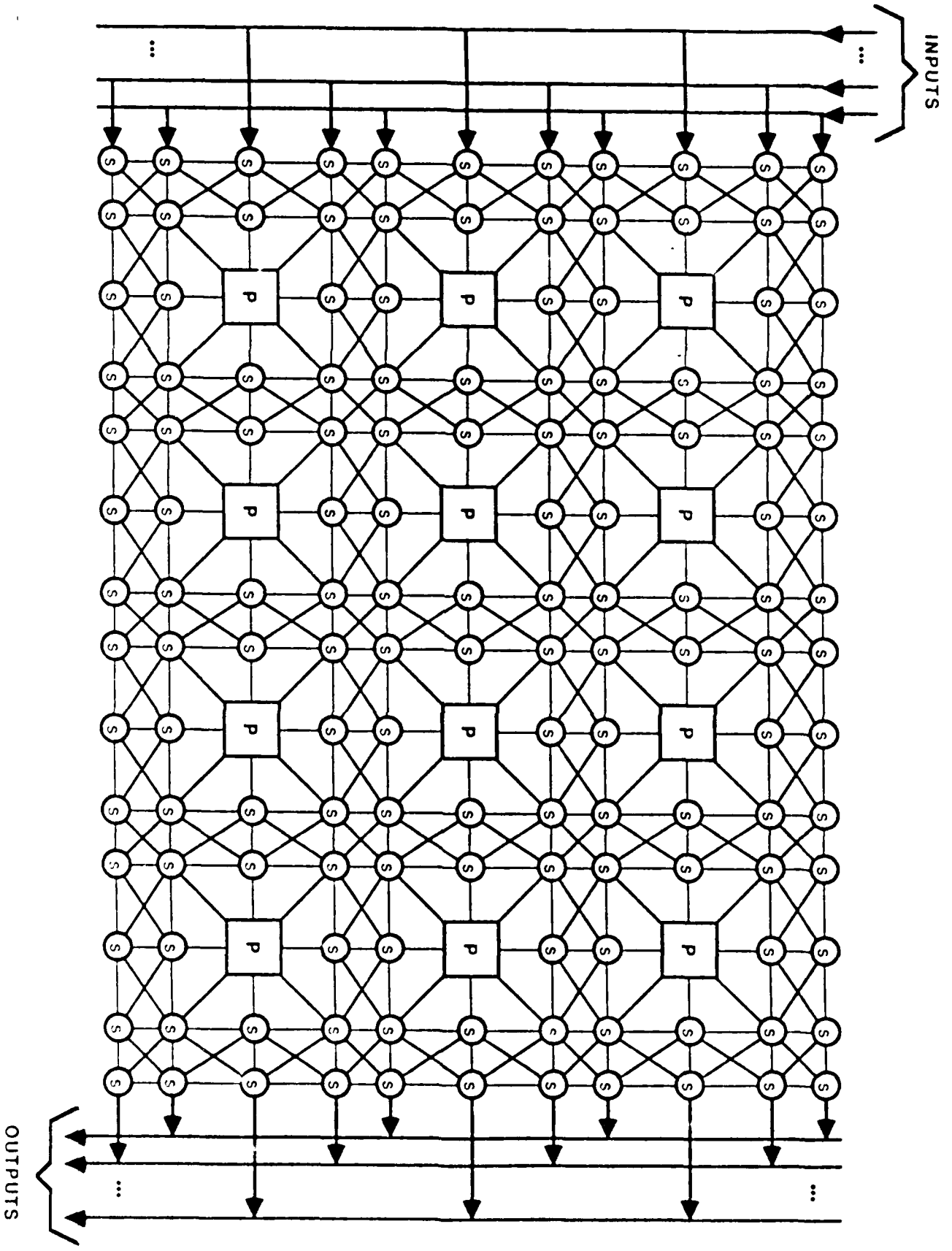


Figure 2. Computational Plane

or S00 at time t it will also change its own state to state A at time t + 1. Any cell in local pattern 000 at time t will stay in local state 0 at time t + 1. Therefore, the result of each cell at time t = 0 observing its local pattern and changing to the appropriate next state will transform the *global pattern*

```
0  0  0  0  0  0  0  S  0  0  0  0  0  0  0
```

at time t = 0, into the global pattern

```
...0  0  0  0  0  0  A  A  A  0  0  0  0  0  0...
```

at time t = 1. In like manner, the global pattern at t = 1 will be transformed into the global pattern

```
...0  0  0  0  0  B  B  C  B  B  0  0  0  0  0...
```

at t = 2 by each cell applying the following local transformation

local pattern			next state
0	0	0	0
0	0	A	B
0	A	A	B
A	A	A	C
A	A	0	B
A	0	0	B

Eventually, the desired global pattern "COMPUTERS" will be reached and will remain stable if each cell applies the following local transformation.

0	0	C	0
0	C	O	C
C	O	M	O
O	M	P	M
M	P	U	P
P	U	T	U
U	T	E	T
T	E	R	E

In general, any such control hyperplane can be characterized as a tessellation automaton [3].

The *tessellation automaton* (TA) is a four tuple

```
TA=(A,Ed,X,c)
```

where,

1. A is a finite non-empty set called the *state alphabet*. For our previous example, A = {0,S,A,B,C,I,E,O,M,P,U,T,R}

2. Ed is the set of all d-tuples of integers called the *tessellation space*. Here the tessellation space is said to be "d" dimensional. In our example, Ed is simply the set of all integers

   ```
   ... -3, -2, -1, 0, 1, 2, 3, ...
   ```

   Ed defines the spatial location of each cell in the array.

3. X is an n-tuple of distinct d-tuples of integers called the *neighborhood index*. Each cell is said to have n neighbors and n is called the *neighborhood scope*. In the example, n = 3, and X = (-1, 0, 1). X defines the relative coordinates of a cells neighbors. For, example, cell 5 has

neighbors (4,5,6) obtained by adding each coordinate of the neighborhood index to the cell location (5).

4. σ is a mapping from An into A called the *local transformation.*

Each cell will decide its next state by observing the present state of its neighbors. In the example, σ(BCB) = I, σ(MPU) = P, etc. It is desirable from the stability point of view that each cell be its own neighbor [4].

For our work we can conveniently characterize the control hyperplane as a two dimensional tessellation automaton, since the computation hyperplane is a two dimensional array of switches and processing elements. To completely describe the control hyperplane, we need to specify the neighborhood index and the local transformation σ. The only difference from the example is that the neighborhood index and the local transformation will be two dimensional in nature.

During the course of this research program, this architecture was studied extensively. Theoretical issues related to automatic reconfiguration in the presence of faults were investigated. Complexity issues were addressed. Problems related to the reliable transmission of data into and out of the system were studied. Simulation facilities were developed to verify the theoretical results, and finally, a proof of concept example was created to determine the complexity of control cells (in terms of memory requirements) and the complexity of switch cells.

The major results of these investigations are summarized in the following sections.

1.3 References

1. Kung, H.T., "Why Systolic Architectures?" *Computer*, January 1982, pp 37-46.

2. Snyder, L., "Introduction to the Configurable, Highly Parallel Computer", *Computer*, January 1982 pp 47-56.

3. Yamada, H., and Amoroso, S., "Tesselation Automata", *Information and Control*, 1969.

4. Walters, S.M., *Pattern Synthesis and Perturbation in Tesselation Automata,* Ph.D. Dissertation, Virginia Tech., Jan 1980.

2.0 Summary of Results

This section describes the most important results of the research. The material presented here is intended only as a summary of results. For more detailed treatments, see the published papers or interim reports that are referenced at appropriate points.

2.1 Computation Hyperplane

The computation hyperplane is a cellular array of switches and computing elements as shown in Figure 2 on page 4. This section describes a systematic means for specifying the system parameters needed to implement a set of specific architectures in a single reconfigurable system.

The square cells are computing elements, whose design will depend upon the exact computations to be performed by the system. The computing elements must be capable of performing all of the system functions in each of the configurations. It must also be possible to select among the various functions in order to provide dynamic reconfiguration. The design of the computing cells is not considered in this research effort.

The switches are represented by circles. The switch parameters that are most important in determining the complexity of the switch nodes are the number of incident data paths, the number of wires in each data path, and the maximum number of simultaneous data path connections to be made by any one switch.

The number of wires in each data path depends upon the needs of the specific computation. To a limited extent the number of wires can be traded off against the time required to communicate data to the neighbors. Serial data transmission would require only one wire, while completely parallel data transmission would require a logarithmic number of wires. Since this decision is strongly computation dependent, we will not consider it further.

The number of incident data paths must be chosen to provide sufficient flexibility to implement a large number of different configurations. This parameter can be traded off against the maximum number of simultaneous connections permitted. The tradeoff involves the control plane complexity as well as the computation plane complexity.

The number of states required in the control plane to represent all of the different possible connections in the computation plane, (where d is the number of data paths in the computation plane, g is the maximum number of simultaneous connections to be implemented in the computation plane, C(x,y) is the number of combinations of x things taken y at a time, and x! is factorial of x) is given by

$$C(d,0)+[C(d,2)*C(2,2)]/1!+[C(d,4)*C(4,2)*C(2,2)]/2!+\ldots$$
$$+[C(d,2g)*C(2g,2)*C(2g-2,2)*C(2g-4,2)*\ldots*C(4,2)*C(2,2)]/g!.$$

For a proof, see [6] or [15] in the project publications list. The number of states required, and the corresponding number of bits required, for various values of d and g are given in Figure 3 on page 8.

Number of Neighbors in Computation Plane	Number of States/Bits Required in Control Plane			
	g=4	g=3	g=2	g=1
10	8551/14	3826/12	676/10	46/6
8	869/10	764/10	344/9	29/5
6	-	77/7	61/6	16/4
4	-	-	10/4	7/3

Figure 3. Number of States/Bits Required in Control Plane

The number of connections required in the control plane per switch as a function of the number of states and the number of neighbors is shown in Figure 4 on page 9.

Number of States for Switch	Number of Bits Required	Number of Connections Required per Control Cell	
		k=5	k=9
4<S<9	3	15	27
8<S<17	4	20	36
16<S<33	5	25	45
32<S<65	6	30	54
64<S<129	7	35	63
128<S<257	8	40	72
256<S<513	9	45	81
512<S<1025	10	50	90
1024<S<2049	11	55	99
2048<S<4097	12	60	108

Figure 4. Number of Control Plane Pins Required

For a particular technology, the density of connections allowed will restrict the switch design space. For example, if the maximum number of connections per module were 45, then the switch design space would be limited to 512 states for k = 5 (k = number of neighbors in the control plane). This would restrict the design to (d = 10 and g = 1) or (d = 8 and g = 2) or (d = 6 and g = 3), etc. In [6] of the project publications list, it is shown that d = 8 with g = 2 is a good compromise for current technology.

2.2 *Control Hyperplane*

As shown in the introduction, the control hyperplane can be modeled as a tessellation automaton. Since the computation plane is a two dimensional array, the natural representation for the control plane will be a two dimensional tessellation automaton. In order to reduce the interconnection complexity as much as possible and to reduce the amount of memory required, it is essential to reduce the number of neighbors to a minimum. Since the cell must always be its own neighbor, the smallest possible neighborhood size for a two dimensional array is 5. This is called the Von Neumann neighborhood [1]. We will show that this neighborhood is sufficiently complex to control the processes of automatic reconfiguration.

2.2.1 Patterns and Growth

It has been established that control patterns in which all subpatterns, equal in size to the neighborhood index, are distinct are needed to support growth in tessellation automata [2]. Walters [2] also showed a method for constructing such patterns in one and two dimensions. The Von Neumann neighborhood requires a modification of Walters two dimensional patterns. An (1 ,3) pattern in one dimension is a string of length L in which all subpatterns of size three are distinct. If this string is arranged in two dimensions as shown in Figure 5 on page 11, then all internal Von Neumann patterns will also be distinct. To insure distinct patterns at the boundaries, the (1,3) pattern must be modified as follows.

A pyramid is constructed as follows.

```
                    a1
                 a2 a3 a2
              a2 a1 a3 a4 a3
           a3 a1 a4 a2 a4 a5 a4
                    .
                    .
                    .
  ai a1 a(i+1) a2 a(i+1) a3 a(i+1) ... a(i-1) a(i+1) a(i+2) a(i+1)
                    .
                    .
                    .
```

If the number of symbols to be included in the pattern is j, then the above construction should be performed until i = j-2. This forms the upper half of the pyramid. The lower half is obtained by reflecting the upper half about the bottom line. The complete construction for j = 6 is shown in Figure 6 on page 12.
See [6] and [9] in the publications list for the project for details.

The two dimensional growth process is illustrated in Figure 7 on page 13. The two dimensional transformation

```
    a1
a2  a3  a4   ->   a6
    a5
```

which denotes the situation where a cell in state a3 sees its neighbors in states a1,a2,a4,and a5 and goes to the next state of a6, will be represented for convenience in the form

```
a1 a2 a3 a4 a5 -> a6 .
```

The local transformation that produces the growth indicated in Figure 7 on page 13 is shown in Figure 8 on page 14.

At a time t-1 after a seed cell is placed in an initial all-zero configuration, the total number of cells initialized will be $2t^2-2t+1$.

2.2.1.1 Proof of Concept Example

To verify that the complexity of the control hyperplane is reasonable for real architectures, a proof of concept example was designed. Four recently proposed research architectures were selected.

1. Two level banyan network [4].
2. Hyper Tree [5].

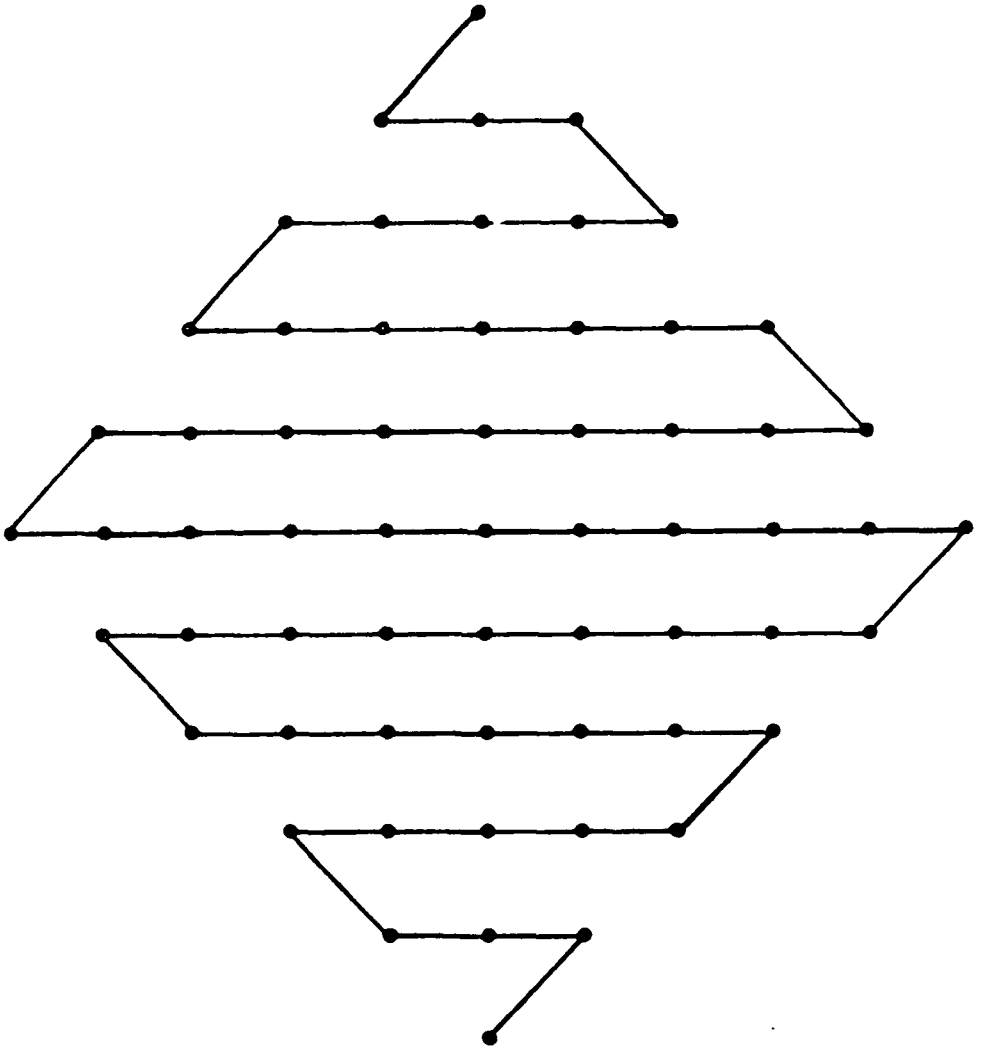


Figure 5. An (L,3) String Arranged in Two Dimensions

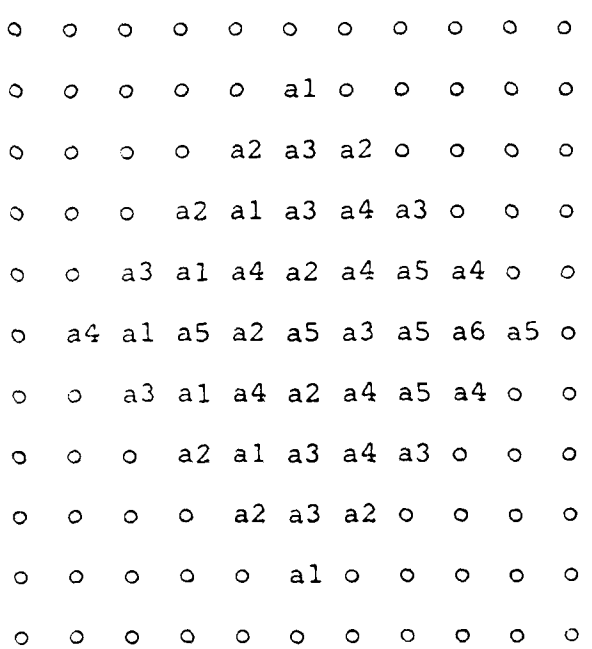


Figure 6. **Two Dimensional Pattern:** All Von Neumann Neighborhoods are Distinct

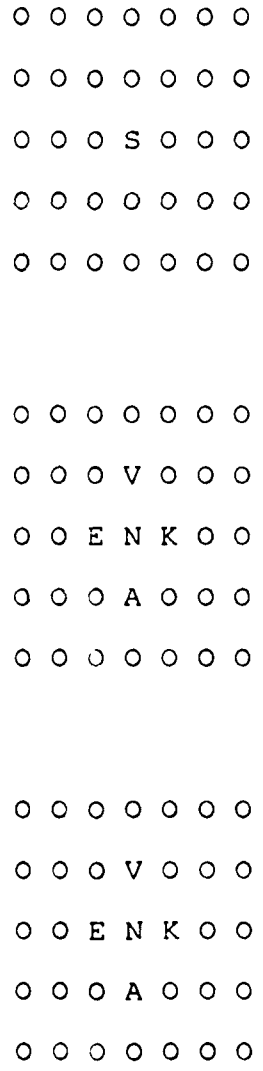


Figure 7. Two Dimensional Growth

O O O O O → O

O O O S O → E

O O O O S → V

O O S O O → N

O S O O O → K

S O O O O → A

O O O O V → O

O O O V E → O

O O O E O → O

E O O A O → O

A O O O O → O

K A O O O → O

O K O O O → O

O V O O K → O

O O V O N → V

V E N K A → N

O O E N O → E

O N K O O → K

N O A O O → A

Figure 8. Local Transformation for Two Dimensional Growth

3. A Fault Tolerant Architecture [6].

4. Lens Structure [7].

The embeddings of these architectures into the computation plane are shown, respectively, in Figure 10 on page 16, Figure 11 on page 17, Figure 12 on page 18, and Figure 13 on page 19. The number of states needed in the control plane cells, along with the number of local transformations for each architecture is shown in Figure 9.

Network Type	Number of Intermediate States	Number of Local Transformations
Banyan Network	52	978
Hyper Tree	42	794
Fault Tolerant Architecture	52	1000
Lens Strategy	63	1232

Figure 9. Memory Requirements for Control Cell Implementation

The total number of states is 209 and the total number of transformations is 4004. The control cell would therefore require an associative memory with 4K 8-bit words. This is within current technology limits.

2.2.1.2 Alignment of Master Pattern

The master pattern may not be aligned properly so that control cells for computations fall over computation cells and switch states fall over switch cells. Algorithms to solve this problem are in reference [6] of the project publications list. Of course, if each cell is capable of both computation and switching, the preferable arrangement, there is no alignment problem.

2.2.1.3 Clearing the Array

The array pattern currently in the system needs to be cleared before a new pattern can be grown. This is done by placing a clear seed in the existing pattern. Any cell that has a neighbor in the clear state will go to the clear state at the next time step. Any cell that is in the clear state will go to the quiscent (or zero) state at the next time period.

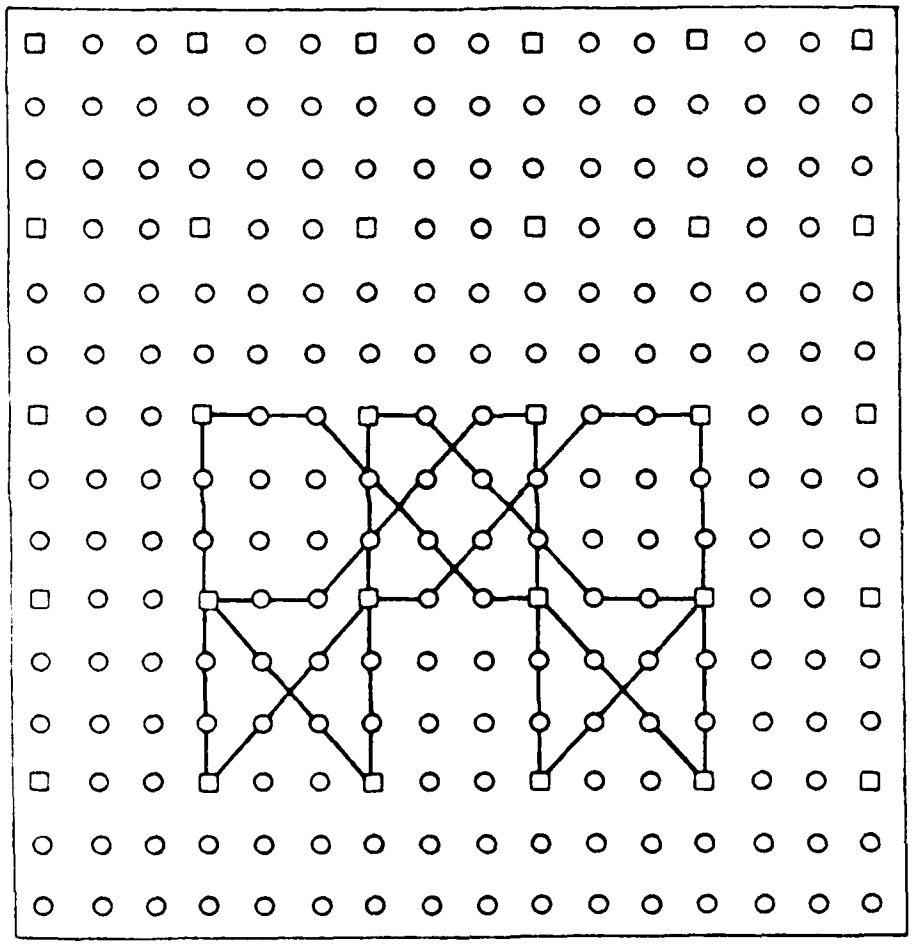


Figure 10. Embedded Two Level Banyan Network

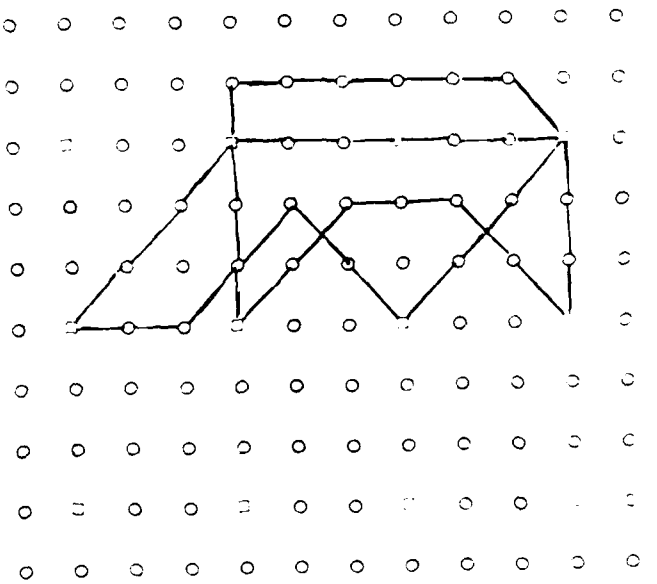


Figure 11. Embedded Hyper Tree

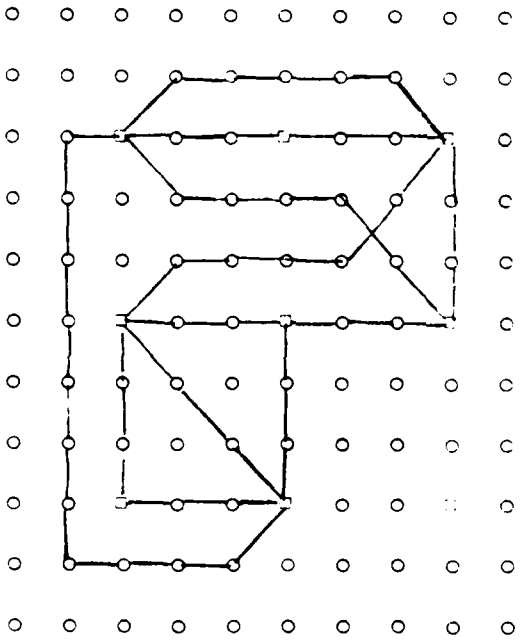


Figure 12. Embedded Fault Tolerant Architecture

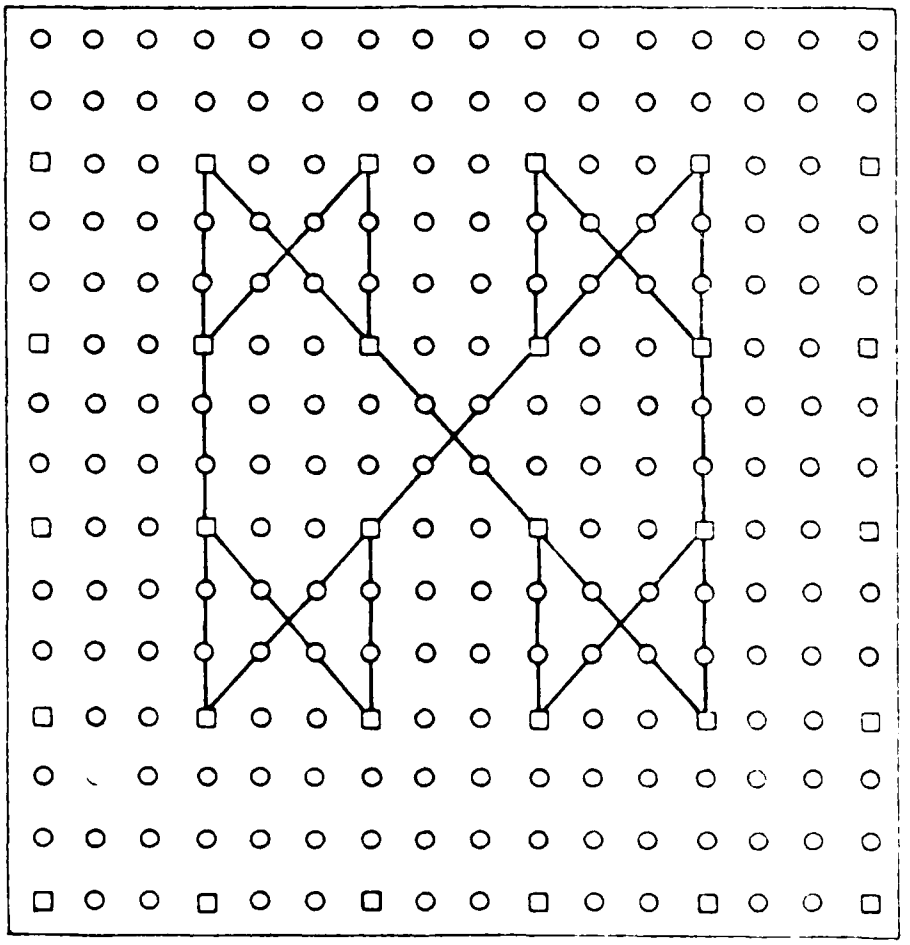


Figure 13. Embedded Lens Network

2.2.2 Fault Tolerance

In addition to having the capability to reconfigure at the request of the user, our system can reconfigure when a fault is detected. The basic approach is to quarantine faulty cells, rather than to report their status to a global interpretation mechanism. Each cell periodically performs tests on its neighbors to determine its opinion of their status. A cell sets its control state to the quarantine state, Q, if it decides that any of its neighbors are faulty. In addition, the cell ignores any inputs from the suspected faulty cells. As a consequence, walls of quarantine states will completely surround regions of faulty cells. Faulty islands will appear in the array that have boundaries of control cells in the Q state.

2.2.2.1 Determining the Size of Fault Free Space

To reconstruct the pattern that existed prior to the detection of the fault, information about the nature of the pattern is extracted from the remaining fault-free cells in the pattern. This information is gathered into a single state, called the *seed state* . This state is ejected into the fault free regions of the array, and searches for an appropriate place to begin growth of the new pattern. Prior to the ejection of the seed, each fault free cell needs to determine the size of the fault-free region that surrounds the cell. This determination is done in an iterative manner, by having each cell examine the condition of its neighbors. After a time, equilibrium will be reached, with each cell knowing the size of the fault-free region that surrounds it.

Each cell is assigned an *s-value* according to the following rules.

1. Each cell in the quarantine state has an s-value of -1.
2. Assuming no wrap-around, each boundary cell of the array that is not part of a quarantine wall will have s-value 0.
3. S-values of all other cells are updated continuously. The s-value of each cell at time i + 1 is the minimum value of the s-values of the set of neighbors of the cell at time i (not including itself).

The s-values are stored in a register called SVR. An example of the s-value computation is shown in the following example for a VonNeumann neighborhood. The distribution of faulty cells and the quarantine wall is shown on the left, with Q indicating a quarantine cell and X indicating a failed cell. On the right is shown the steady state s-value distribution, where t = -1.

```
---QQXQ    000ttXt
--QXXXQ    00tXXXt
---QXQ-    010tXt0
----Q--    0110t00
-------    0121010
-------    0111110
-------    000000t
```

In references [8] and [14] of the project publication list, it is shown that the distribution of s-values in any configuration will reach equilibrium in at most 2L-1 time periods following the occurence of a new fault, where L is the maximum dimension of the rectangular array.

The traveling seed must use the s-value of a cell to determine whether there are enough surrounding fault-free cells in which to grow the desired pattern. In a Moore neighborhood of a d-dimensional array, if the s-value of a cell is x, then the cell is at the geometric center of a hypercube of size (2x + 1). In a Von Neumann neighborhood, if the s-value of a cell is x, then the cell is at the geometric center of a hyperdiamond of girth (2x + 1). In the example above, the cell with s-value = 2 is at the geometric center of a hyperdiamond of girth 5.

2.2.2.2 *Two Dimensional Reconfiguration Algorithm*

The following set of algorithms implement the steps in reconfiguration.

1. Algorithm that coordinates and multiplexes the transmission of information between cells in the control hyperplane.
2. Algorithm to control a reconfiguration source.
3. Algorithm to reduce the number of reconfiguration sources to one.
4. Algorithm to clear the state registers in the fault-free region prior to pattern growth.
5. Algorithm to transfer the seed state from a reconfiguration source into the fault-free region.
6. Algorithm to steer the seed to an appropriate place in the array to start growth.
7. Algorithm to handle collisions between growing patterns so that only one pattern is eventually grown in the array.

Multiplexing Algorithm: The multiplexing is accomplished by partitioning the cells of the control hyperplane into two sets according to a checkerboard pattern. "Red" cells would communicate s-values during a given time period, while "black" cells would communicate state information. During the next time period, roles would be reversed.

The sequencing through reconfiguration steps is accomplished by a finite-state machine in the control cell. This machine needs only six states and two timers.

Neutralization is the name given to the algorithm that reduces the number of possible seed sources to one. An arbitrary unique priority value is assigned to each cell in the array. An example follows.

1	2	3	4	5	6	7	8	9
18	17	16	15	14	13	12	11	10
19	20	21	22	23	24	25	26	27
59	60	61	62	63	31	30	29	28
64	65	66	67	68	32	44	43	58
79	78	77	72	69	33	45	42	57
0	73	76	71	39	34	40	41	56
80	74	75	70	38	35	46	49	50
55	54	53	52	37	36	47	48	51

The only requirement is that each priority value be unique. When a cell goes into the quarantine state, it becomes a possible candidate for becoming the seed source. It sends its priority value to each of its neighboring cells. After a time, the potential reconfiguration source with the largest priority value will become the single seed source. In the following example, after faults are detected, x indicates a failed cell, W indicates a quarantine state in the quiescent region, V indicates a antine state is the active region (these are the potential sources), 0 indicates a quiscent cell, and a lower case letter indicates the other active cells.

```
0 0 0 0 0 0 0 0 0
0 0 0 0 0 W 0 0 0
0 0 0 0 W x W 0 0
a b c a V x W 0 0
c f d b e V 0 0 0
d V c d f a 0 0 0
V x V e d b 0 0 0
c V e d a f 0 0 0
e c d f c d 0 0 0
```

After a state change, the global pattern becomes as follows.

```
0  0  0  0  0  0  0  0  0
0  0  0  0  0  W  0  0  0
0  0  0  0  W  x  W  0  0
u  u  u  u 63  x  W  0  0
u  u  u  u  u 32  0  0  0
s 78  s  s  s  s  0  0  0
0  x 76  u  u  u  0  0  0
u 74  u  u  u  u  0  0  0
u  u  u  u  u  u  0  0  0
```

The array then goes through the sequence of steps shown in Figure 14 on page 23 and Figure 15 on page 24.
In these figures, + indicates a neutralized cell. In the last step, the only remaining potential source is desginated V.

The clearing step is accomplished by allowing all cells with 78 codes to go to the 0 state. The result is shown in Figure 16 on page 25.

```
 0  0  0  0  0  0  0  0  0
 0  0  0  0  0  W  0  0  0
 0  0  0  0  W  x  W  0  0
 u  u  u 63 63  x  W  0  0
 u 78  u  u 63 32 32  0  0
78 78 78  s  s 32  0  0  0
 0  x 76 76  u  u  0  0  0
74 74 76  u  u  u  0  0  0
 u 74  u  u  u  u  0  0  0
```

```
 0  0  0  0  0  0  0  0  0
 0  0  0 63  0  W  0  0  0
 0 78 63 63  W  x  W  0  0
78 78 78 63 63  x  W 32  0
78 78 78 78 63  +  63 32 32
78 78 78 78 78 63 32 32  0
 +  x  + 78 63 32 32  0  0
78 76 78 76 76 32  0  0  0
74 76 76 76  u  u  0  0  0
```

```
 0  0  0  0  0  0  0  0  0
 0  0  0  0  0  W  0  0  0
 0  0  0 63  W  x  W  0  0
 u 78 63 63 63  x  W  0  0
78 78 78 63 63  +  32 32  0
78 78 78 78 63 32 32  0  0
 +  x  + 76 32 32  0  0  0
74 76 76 76  u  u  0  0  0
74 74 76  u  u  u  0  0  0
```

```
 0  0  0 63  0  0  0  0  0
 0 78 63 63 63  W  0  0  0
78 78 78 63  W  x  W 32  0
78 78 78 78 63  x  W 32  0
78 78 78 78 78  +  63 63 32
78 78 78 78 78 78 63 32 32
 +  x  + 78 78 63 32 32  0
78  +  78 78 76 76 32  0  0
78 76 78 76 76 32  0  0  0
```

Figure 14. Steps 1-4

```
 0  0  0 63  0  0  0  0  0
 0 78 63 63 63  W  0  0  0
78 78 78 63  W  x  W 32  0
78 78 78 78 63  x  W 32  0
78 78 78 78 78  + 63 63 32
78 78 78 78 78 78 63 32 32
 +  x  + 78 78 63 32 32  0
78  + 78 78 76 76 32  0  0
78 76 78 76 76 32  0  0  0
```

```
78 78 78 78 78 78 78 78 78
78 78 78 78 78  W 78 78 78
78 78 78 78  W  x  W 78 78
78 78 78 78  +  x  W 78 78
78 78 78 78 78  + 78 78 78
78 78 78 78 78 78 78 78 78
 +  x  + 78 78 78 78 78 78
78  + 78 78 78 78 78 78 78
78 78 78 78 78 78 78 78 78
```

```
 0 78 63 63 63  0  0  0  0
78 78 78 63 63  W  0 32  0
78 78 78 78  W  x  W 32 32
78 78 78 78  +  x  W 63 32
78 78 78 78 78  + 63 63 63
78 78 78 78 78 78 78 63 32
 +  x  + 78 78 78 63 32 32
78  + 78 78 78 76 76 32  0
78 78 78 78 76 76 32  0  0
```

```
78 78 78 78 78 78 78 78 78
78 78 78 78 78  W 78 78 78
78 78 78 78  W  x  W 78 78
78 78 78 78  W  x  W 78 78
78 78 78 78 78  W 78 78 78
78  V 78 78 78 78 78 78 78
 W  x  W 78 78 78 78 78 78
78  W 78 78 78 78 78 78 78
78 78 78 78 78 78 78 78 78
```

Figure 15. Steps 5-8

```
O  O  O  O  O  O  O  O  O
O  O  O  O  O  Y  O  O  O
O  O  O  O  Y  x  Y  O  O
O  O  O  O  Y  x  Y  O  O
O  O  O  O  O  Y  O  O  O
O 'Y  O  O  O  O  O  O  O
Y  x  Y  O  O  O  O  O  O
O  Y  O  O  O  O  O  O  O
O  O  O  O  O  O  O  O  O
```

Figure 16. The array after clearing.

In this figure, Y indicates normal quarantine cells, and 'Y indicates the unique seed source.

The cell indicated by 'Y then ejects a seed into the fault-free region. This seed then migrates to an appropriate place in the fault-free region, indicated by the s-values of cells. This migration can handle collisions between growing patterns and newly arrived faults. All of these algorithms are described in detail in references [8] and [13] in the publications list for the project.

To determine the complexity of the control logic, a sample design was constructed. Systolic arrays were implemented to multiply bandwidth three matrices, and to compute the LU decomposition of bandwidth three matrices. The design required 35 states with only 30 mapping points in the local transformation function. Thus, the control memory is very small. In addition, a total of 94 bits of control status information must be maintained (for s-values, state registers, flags, etc.). Information is transmitted in two successive four bit nibbles with two control lines for synchronization. Since the Von Neumann neighborhood is being used, this requires 24 connections to each cell (six from each neighbor). The worst case reconfiguration time after fault detection is 4096 clock periods (about 0.8 milliseconds at a clock frequency of 5 MHz). With a 3x3 array being embedded in an 8x8 array, the system could reconfigure a maximum of 25 times before running out of fault-free cells.

2.2.2.3 One Dimensional Reconfiguration Algorithm

One-Dimensional arrays are very restrictive in terms of the paths available for control information flow. Therefore, special considerations are necessary. The multi-dimensional study applies to all arrays with dimension greater than one.

Several additional data paths must be added for reconfiguration. Several additional flags and algorithms must also be added. For a complete description of these additions, see [5], [8] and [12] in the project publications list.

2.2.3 Input/Output Path Construction

The active array that is embedded within a larger array must communicate results to the outside world, and must receive data from the outside world. In a reconfigurable system, the position of the active cells will change as a result of reconfiguration. This means that the data paths to the

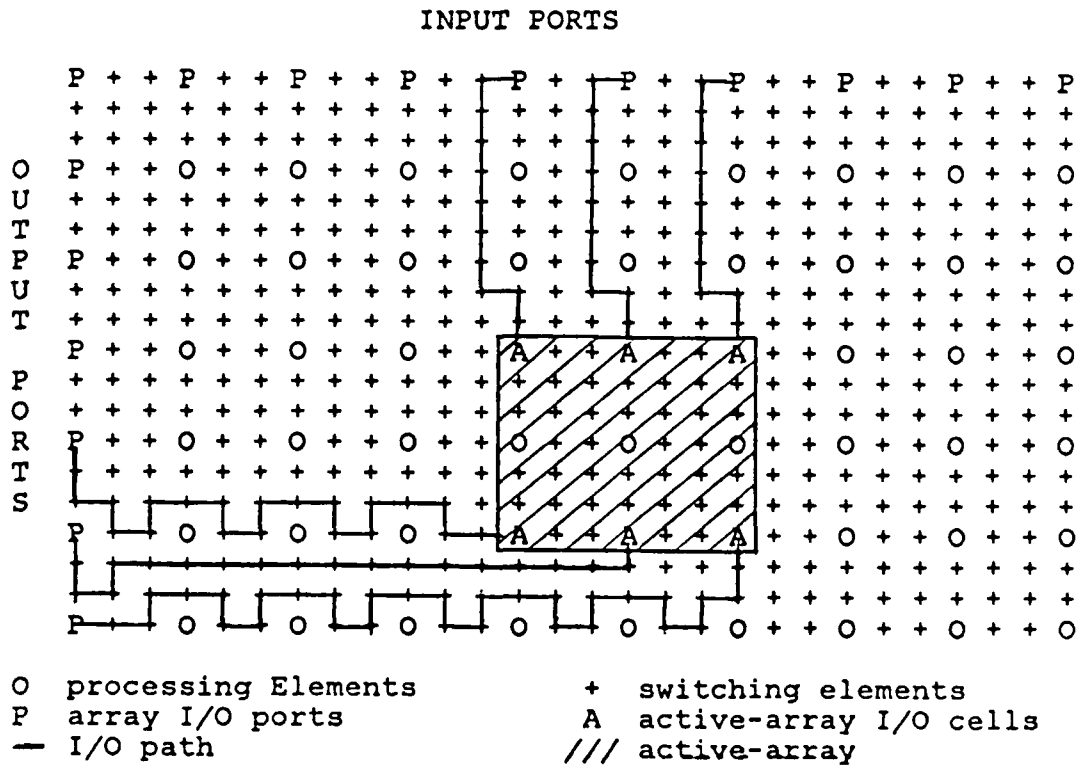


Figure 17. Input/Output Path Construction

outside world must also change. In addition, if a node in the data path fails, a new data path must be provided.

There are three types of elements in a data path. The first type is the array I/O port that must exist on the array boundary and is connected directly to the outside world. This type will be called an external I/O port and will be designated by the symbol P. The second type, called an internal I/O port and designated by A, is the node in the active array that receives data from the external I/O port or supplies output data for the external I/O port. The third type is an element in the data path that connects an external I/O port to an internal I/O port. These elements are illustrated in Figure 17 on page 26.

The external I/O ports must be on the boundary of the array. In Figure 17 on page 26 the external input ports are located along the top edge of the array, and the external output ports are located along the left edge of the array. The data paths connect these external ports with the internal I/O ports. The active array is shaded.

Since the location of an external I/O port may change due to reconfiguration, there must be some means of identifying the port to the outside world. This is accomplished by adding tag bits to data that passes through the ports. These tag bits identify the port to the outside world.

Cells in the data path are tagged so that if a faulty cell is identified, only path regrowth is initiated. It is not generally necessary to reconfigure the entire array if a path fault occurs. If a cell in some data path enters the quarantine state, it either initiates a "find alternate path" message if it is between the faulty cell and an internal I/O port, or initiates a clear path function if it is between the faulty cell and an external I/O port. When the clear message reaches the external I/O port, it stops sending or receiving data.

The internal I/O ports must initiate I/O path growth during system initialization or when active array reconfiguration occurs. The system designer should insure that the internal I/O ports are on active array edges that "face" toward the corresponding external I/O ports; otherwise, the growth of I/O paths will block each other.

The path is grown by passing a pointer tag from an internal I/O port through the intermediate cells to an external I/O port. The cell with the pointer tag queries its neighbors to determine their status. Based on the responses, and using a system of priorities, the pointer decides in which direction to pass the pointer tag. If a path becomes blocked, the cell with the pointer tag enters a "backtrack" state, and passes the pointer tag back to the predecessor cell in the path. In references [10] and [16] in the project publication list, it is proved that the algorithm will find a path if it exists, and that path collisions will be effectively overcome. An example is shown in Figure 18 on page 28.

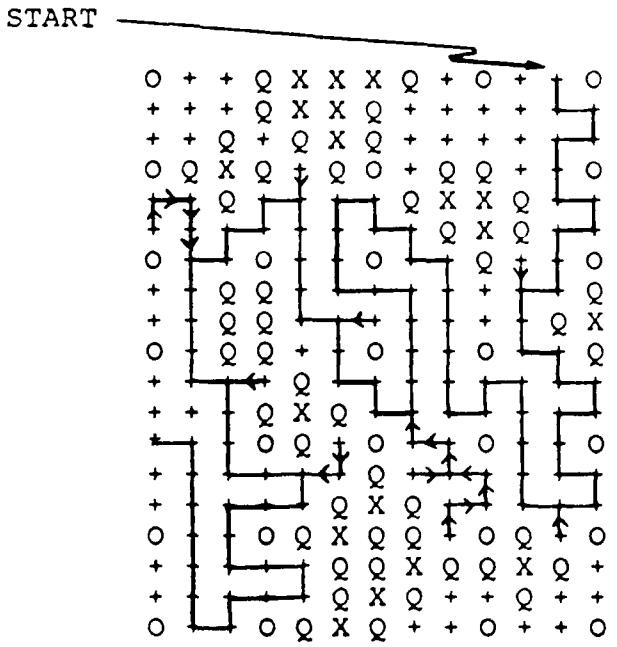
2.3 *System Concepts*

In this section, concepts that pertain to the entire array are considered.

2.3.1 Synchronized Clocks

To work correctly, all cells in the array must be clocked in exact synchronism. During this project, we developed a clocking system that employs $2f + 2$ modules in order to tolerate up to f module failures. All remaining good clock modules will maintain a synchronized clock as long as no more than f modules have failed. This work is described in publication [4] of the project publication list.

For systems that are too large to support the number of interconnections required by the method described above, we have developed an alternative synchronization method described in publication [17]. This method is based on the concept of overlapping cycles.



O processing elements
X faulty cells
\+ switching elements
* only available I/O port

Q cells in quarantine
-- path
->- path with backtrack

DIRECTION PRIORITY IS NESW

Figure 18. Example of I/O Path Growth

2.3.2 Simulation

A complete simulation package was developed in which the reconfiguration algorithms could be tested. This work will appear in publication [11] which was in progress when this report was prepared. Using the simulator, several improvements to the growth algorithms were discovered. These will also be reported at a later date.

2.3.3 Periodic Self Restoration

This part of the research is directed at possible redundancy techniques that could be used to achieve fault tolerance. It is a study of an architecture that might be embedded in the reconfigurable system described in the previous sections. In this sense, the purpose is quite different from that of earlier sections.

The dynamic redundancy methods that are widely used in fault tolerant systems can be considered to be event driven. In such systems, it is the detection of an error that triggers the attempt to restore the system to a correct operation state. In the proposed periodically self restoring redundancy (PSRR) scheme, an alternative approach is presented in which the system periodically restores itself (whether or not a fault has occurred) so as to correct any errors before they build up to the point of system failure.

The PSRR scheme employs N computing units operating redundantly in tight synchronization. Each computing element has full functional capabilities and could, if necessary, perform all system functions on its own. System input must be supplied to each computing element, and system output is computed according to decision rules that are described in publication [2] of the project publication list. If the system is operational, the consensus output is guaranteed to be error free.

The failures may be due to either permanent or temporary faults. While computing units that have permanent faults cannot be correctly restored, those that have transient faults can be reliably resynchronized with the rest of the system. To achieve this, N computing elements periodically communicate their state information to each other and resynchronize themselves to a mutual consensus state. If a minimum number of computing elements are operational, this consensus is assured of being the correct operational state. The restoration is initiated by a non-maskable interrupt from a fault tolerant clock and is executed out of ROM. This ensures that a computing element that has failed due to a transient error will execute the restoration algorithm. It is therefore restored to the operational state if enough other computing elements in the system are operational.

The PSRR system is particularly effective at handling transient faults. This is an important advantage because transient faults are believed to occur much more frequently than permanent faults. PSRR systems may also tolerate a limited number of permanent faults due to the consensus operation.

A fixed computation-restoration cycle is established. This consists of a computing interval followed by a restoration interval during which the system is restored. Shorter CR cycle times imply more frequent restoration and hence more reliable operation. The price is lowered throughput since the restoration time is fixed.

During this study, this system was modeled using a Markov model. Procedures for determining the system reliability and the system mean time to failure are developed in terms of the individual computing element failure probabilities for permanent and transient faults during a CR cycle, the CR cycle time, and the mission time. It is found that the reliability can be increased either by more frequent restoration, which degrades throughput, or by increasing the level of redundancy, which increases cost. A method is developed that allows evaluation of the level of redundancy needed for a given CPU to meet desired performance and reliability specifications. If the procedure is used on all available CPU's the best choice for given specifications can be determined.

The PSSR scheme allows design optimization based on parameters that can be estimated with reasonable accuracy at the specification level. This is in contrast to most present day fault tolerant systems in which the coverage factors for test procedures are virtually impossible to predict in ad-

vance of system implementation. Another advantage is that PSRR systems are realized with off the shelf components.

The details of this system are published in publications [2] and [18] in the project publications list.

2.3.4 Cell Testing Techniques

Although cell testing was not a primary focus of this research effort, some initial investigation into this problem was done. The types of test to be performed were determined as well as a preferred order. Preliminary work on partitioning the test set for sequential passes was done. Suggestions for possible algorithms also were made. This work is only preliminary in nature; much remains to be done. The results to date can be found in publication [10] in the projects publication list.

2.4 References

1. J. VonNeumann, *Theory of Self-Reproducing Automata*, University of Illinois Press, Urbana, Illinois, 1966.

2. S.M. Walters, *Pattern Synthesis and Perturbation in Tessellation Automata*, Ph.D. Dissertation, Virginia Polytechnic Institute and State University, January 1980, 250 pages.

3. I.J. Good, "Normal Recurring Decimals", *Journal London Mathematical Society*, Vol. 21, 1946, pp. 167-169.

4. R.I. Goke, *Banyan Networks for Partitioning Multiprocessor Systems*, Ph.D. Thesis, University of Florida, 1976.

5. J.R. Goodman and C.H. Sequin, "Hypertree: A Multiprocessor Interconnection Topology", *IEEE Transactions on Computers*, Vol C-30, December 1981, pp. 923-933.

6. D.K. Pradhan and S.M. Reddy, "A Fault Tolernat Communication Architecture for Distributed Systems", *IEEE Transactions on Computers*, Vol. C-31, September 1982, pp. 863-869.

7. R.A. Finkle and M.H. Solomon, "The Lens Interconnection Strategy", hp1.IEEE Transactions on Computers, Vol. C-30, December 1981, pp. 960-965.

3.0 Publications and Technical Reports

1. F.G. Gray, "General Purpose Reconfigurable Architecture", *Proceedings of the 1982 International Conference on Circuits and Computers*, New York, New York, September 28 - October 1, 1982, pp. 122-125.

2. A.D. Singh and F.G. Gray, *The Design of Periodically Self Restoring Redundant Systems*, Ph. D. Dissertation, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, December 1982, 127 pages. Also, Interim Technical Report No. 1, Army Research Office, DAAG29-82-K-102, February 1983.

3. R. Kumar and F.G. Gray, "Control Patterns in Cellular Arrays", *Proceedings of SouthEastcon84*, Louisville, Kentucky, April 8-11, 1984, pp. 443-448.

4. N. Gollakota and F.G. Gray, "Fault Tolerant Clocks in Arrays of Processors", *Proceedings of SouthEastcon84*, Louisville, Kentucky, April 8-11, 1984, pp. 449-452.

5. R. Kumar and F.G. Gray, "A Fault Tolerant One Dimensional Structure", *The 4th International Conference on Distributed Computing Systems*, May 14-18, 1984, San Francisco, CA., pp. 472-483.

6. Naga Gollakota, *Automatically Reconfigurable Highly Parallel Computer Systems*, Masters Thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, June 1984, 102 pages.

7. R. Kumar and F.G. Gray, "Reconfigurable Cellular Arrays", *Proceedings of the 27th Midwest Symposium on Circuits and Systems*, Morgantown, West Virginia, June 11-12, 1984.

8. Rajesh Kumar, *A Fault Tolerant Cellular Architecture*, Ph.D. Dissertation, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, July 1984, 268 pages.

9. N. Gollakota and F.G. Gray, "Reconfigurable Cellular Architecture", *Proceedings of 1984 International Conference on Parallel Processing*, August 21-24, 1984, Bellaire, Michigan, pp.377-379.

10. Kathleen Connell, *I/O Algorithm and a Test Algorithm for a Reconfigurable Cellular Array*, Master's Thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, June 1985, 168 pages.

11. Bryan Brighton, *Simulation of a Fault Tolerant Parallel Architecture*, Masters Thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, in progress.

12. R. Kumar and F.G. Gray, "A Fault Tolerant One Dimensional Cellular Architecture", in preparation.

13. R. Kumar and F.G. Gray, "A Fault Tolerant Multi-Dimensional Cellular Architecture", in preparation.

14. R. Kumar and F.G. Gray, "The Determination of Fault-Free Spaces in Cellular Arrays", in preparation.

15. N. Gollakota, J.C. McKeeman, and F.G. Gray, "Data Path Reconfiguration in an Array Structure", in preparation.

16. J.C. McKeeman and F.G. Gray, "I/O Algorithm for a Reconfigurable Array Architecture", in preparation.

17. M. Roumeliotis, "An Improved Synchronized Array Clock", in preparation.

18. A.D. Singh and F.G. Gray, "Periodically Self Restoring Redundant Systems for VLSI Based Highly Reliable Design", Proceedings of EUROMICRO84, 11 pages.

4.0 Participating Scientific Personnel

Principal Investigators:

1. Dr. F. Gail Gray -(June 1982 - December 1985)
2. Dr. John C. McKeeman - (June 1984 - December 1985)

Graduate Research Assistants:

1. Adit D. Singh - (June 1982 - December 1982) - Ph.D. awarded December 1982.
2. Rajesh Kumar - (January 1983 - June 1984) - Ph.D. awarded July 1984.
3. Naga Gollakota - (January 1983 - June 1984) - MSEE awarded June 1984.
4. Kathleen Connell - (June 1984 - May 1985) - MSEE awarded July 1985.
5. Bryan Brighton - (September 1984 - May 1985) - MSEE pending
6. Manos Roumeliotis - (Not supported)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ARO 18803.14-EC	2. GOVT ACCESSION NO. N/A	3. RECIPIENT'S CATALOG NUMBER N/A
4. TITLE (and Subtitle) Fault Tolerance in Parallel Architectures		5. TYPE OF REPORT & PERIOD COVERED Final Report June 1982 - Dec. 1985
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Dr. F. Gail Gray		8. CONTRACT OR GRANT NUMBER(s) DAAG 29-82-K-0102
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Electrical Engineering Virginia Polytechnic Inst. and State University Blacksburg, VA 24061		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS U. S. Army Research Office Post Office Box 12211 Research Triangle Park, NC 27709		12. REPORT DATE May 30, 1986 13. NUMBER OF PAGES 33
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

NA

18. SUPPLEMENTARY NOTES

The view, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Fault Tolerance	Distributed Algorithms
Parallel Processing	Computer Architecture
Reconfiguration	Distributed Control

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper describes a proposed automatically reconfigurable cellular architecture. The unique feature of this architecture is that the reconfiguration control is distributed within the system. There is no need for global broadcasting of switch settings. This reduces the interconnection complexity and the length of data paths. The system can reconfigure at the request of the applications software or in response to detected faults. This architecture supports fault tolerant applications since the reconfiguration can be self- triggered from within. The complete reconfiguration process can proceed without external interference.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

END

DTIC

8—86